Sequential Consistency and Concurrent Data Structures

Ali Sezgin[1]

1 **University of Cambridge, UK**
as2418@cam.ac.uk

Abstract

Linearizability, the de facto correctness condition for concurrent data structure implementations, despite its intuitive appeal is known to lead to poor scalability. This disadvantage has led researchers to design scalable data structures satisfying consistency conditions weaker than linearizability. Despite this recent trend, sequential consistency as a strictly weaker consistency condition than linearizability has received no interest.

In this paper, we investigate the applicability of sequential consistency as an alternative correctness criterion for concurrent data structure implementations. Our first finding formally justifies the reluctance in moving towards sequentially consistent data structures: Implementations in which each thread modifies only its thread-local variables are sequentially consistent for various standard data structures such as pools, queues and stacks. We also show that for almost all data structures, and all the data structures we consider in this paper, it is possible to have sequentially consistent behaviors in which a designated thread does not synchronize at all. As a potential remedy, we define a hierarchy of quantitatively strengthened variants of sequential consistency such that the stronger the variant the more synchronization it enforces which at the limit is equal to that enforced by linearizability.

Keywords and phrases Concurrency, Formal Specification, Data Structures, Sequential Consistency, Linearizability

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

The tension between performance and correctness is well known when it comes to developing low-level library routines implementing concurrent data structures [16]. On the one hand, scalability, the ability to fully utilize the parallelism offered by the underlying architecture and generally accepted to be the main determinant of overall performance, is adversely affected by the need to synchronize among threads. On the other hand, correctness criteria, usually known as consistency conditions, enforce lower bounds on the amount of synchronization.

In order to break the impasse in favor of scalability research has focused on weakening the notion of correctness. The move towards alternative consistency conditions potentially leading to better performance was initially in the domain of memory implementations. The goal was to replace sequential consistency [13] (SC) with weaker memory consistency models (e.g. [2, 3, 6, 8, 14]). Recently a similar surge has been going on relative to linearizability [10], which has hitherto been the criterion for correctness in the domain of concurrent data structures, being replaced with weaker notions of correctness (e.g. [5, 9, 11]). There have been already many scalable implementations benefiting from these weaker notions (e.g. [1, 4, 7, 9, 12, 15]).

In this paper, we investigate SC as an alternative relaxation of linearizability. Linearizability requires that the effect of a method be globally visible (i.e. to other executing threads)

Conference title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–13

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

before it completes. SC relaxes this by requiring only thread local ordering. That is, if two methods m and m' are called by the same thread in that order, then the requirement by SC is that the effect of m *appear to* be before that of m'. Furthermore, because SC is generally accepted to be the most intuitive memory consistency model and concomitantly is very well understood and studied, it is surprising that it has received no consideration until now. Intriguing though it may be, we show that the apparent reluctance is actually warranted.

Our investigation of SC ranges over five common data structures: two variants of pools ($\mathcal{P}$, $\mathcal{P}^?$), queues ($\mathcal{Q}$), stacks ($\mathcal{S}$) and register (banks) ($\mathcal{R}$). We show that for all five of them it is possible to construct SC implementations where one thread, say t, can arbitrarily delay its synchronization with other threads as long as the sequence of local events of t satisfy a certain property, which we call *robustness*. Basically a sequence of events, method calls with return values, over a data structure $\mathcal{D}$ is robust if the same sequence can be executed regardless of the state $\mathcal{D}$ is in. For instance, enqueueing or pushing an element is a robust sequence (of length 1) in $\mathcal{Q}$ and $\mathcal{S}$, respectively.

We also show that for pool, queue and stack implementations being SC guarantees even less. We define the class of *composable* data structures to which pool, queue and stack belong. Intuitively a data structure is composable if for any pair of valid behaviors there exists an interleaving of this pair which is again a valid behavior. For composable data structures, an implementation in which threads do not synchronize at all is SC. To better understand the strength of such a result, consider a typical concurrency programming problem which contains N tasks to be generated by the producer threads and to be completed by the consumer threads. If the producers are to convey their tasks to the consumers over an SC queue (pool or stack), due to lack of synchronization the program will end with the queues of producers containing tasks and consumers having done nothing at all!

As a possible remedy, we propose a natural modification to the definition of SC. We call an implementation k-SC if every thread has to synchronize at least once after executing k local events. This definition is strong enough to rule out the pathological implementations we consider in this paper. It also naturally spans the domain of implementations between SC and linearizability via a quantitative stratification. The smaller k is, the stronger k-SC is which at the limit reduces to linearizability (equivalent to 0-SC).

To summarize, we make the following contributions:

- Define the properties robustness and composability for data structures,
- Prove that essentially broken SC implementations exist for data structures with either of these properties,
- Span the range between SC and linearizability by bounding the non-synchronized event sequences.

1.1 Related Work

Directly related to our work, there have been two other work on relaxing the notion of linearizability. In [9], Henzinger et al. propose a framework in which it is possible to quantitatively relax any sequential data structure. Their framework enables one to define a desired metric and for any data structure $\mathcal{D}$ defines $\mathcal{D}_k$ to be all behaviors that are at most k away from some valid behavior of $\mathcal{D}$. A concurrent behavior is associated with the set of potential sequential witnesses, as defined by linearizability, and the concurrent behavior is correct if at least one potential witness belongs to $\mathcal{D}_k$. In contrast, our work modifies the set of potential sequential witnesses, by using SC rather than linearizability, and the concurrent behavior is correct if at least one potential witness from this extended

set belongs to $\mathcal{D}$. In [11], Jagadeesan and Riely consider quiescent consistency (QC) as an alternative relaxation to linearizability. They quantitatively span the range between QC and linearizability. Similar to our work, their quantitative metrics is also defined over concurrent behaviors. Unlike us, they do not consider whether QC allows pathological cases. Neither work formally establishes the relation between synchronization and the characteristics of data structures as we do in this work.

2 Notation

For a set A, let $Pow(A)$ denote the set of all subsets of A. Let $\lambda x.0$ with x ranging over elements of A denote the function that maps all elements of A to 0. Let $A[B]$ denote the collection of functions from A to B. For $f \in A[B]$, $f[a \mapsto b]$ denotes the function that agrees with f on A except for a which is mapped to b. A sequence a of length n over some alphabet A is denoted by $a(1) \cdot a(2) \ldots \cdot a(n)$. Alternatively, we also use the notation $\langle a(i) \rangle_{i \in [1,n]}$ to denote a. Let $len(a)$ denote the length of a. For simplicity of presentation, unless stated explicitly to be otherwise, for every sequence a and for any $1 \leq i, j \leq len(a)$ we assume that $a(i) \neq a(j)$. Let $last(a)$ denote the last symbol of a; i.e. $last(a) = a(len(a))$. Let $Set(a)$ denote the set of symbols occurring in a; i.e. $Set(a) = \{a(i) \mid i \in [1, len(a)]\}$. Each sequence a induces a total order over $Set(a)$, *appears-before* order $<_a$, such that $i < j$ iff $a(i) <_a a(j)$. Let A^* denote the set of all sequences over A, with ε denoting the *empty* sequence.

A labelled transition system (LTS) is a tuple $\mathsf{LTS} = (Q, q_0, L, \rightarrow)$, where Q is the set of *states*, q_0 is the *initial* state, L is a set of labels, and $\rightarrow \subseteq (Q \times L \times Q)$ is a *transition relation*. We write $q \xrightarrow{l} q'$ if $(q, l, q') \in \rightarrow$. A *run* $\mathbf{r} = q_0 \cdot l_1 \cdot q_1 \ldots l_n \cdot q_n$ is an alternating sequence of states and labels such that for all $i \in [1, n]$ we have $q_{i-1} \xrightarrow{l_i} q_i$. The *trace* of $\mathbf{r}$, $tr(\mathbf{r})$, is the sequence of labels occurring in $\mathbf{r}$; i.e. $tr(\mathbf{r}) = \langle l(i) \rangle_{i \in [1,n]}$. Let $\mathsf{Tr}(\mathsf{LTS})$ denote the set of all traces of LTS.

2.1 Data Structures

A *data structure* $\mathcal{D}$ is a pair $(D, \Sigma_{\mathcal{D}})$, where D is the *data domain* and $\Sigma_{\mathcal{D}}$ is the *method alphabet*. For all the data structures we consider in this paper, we take D to be the set of natural numbers, $\mathbb{N}$, possibly augmented with a distinguished symbol `NULL`. An event of $\mathcal{D}$ is a quadruple (id, m, d_i, d_o), where $id \in \mathbb{N}$ is an *event identifier*, $m \in \Sigma_{\mathcal{D}}$ is a method, $d_i, d_o \in D$ are *input* and *output* arguments, respectively. Intuitively, (id, m, d_i, d_o) denotes the application of method m with input argument d_i returning the output value d_o. When the input (resp. output) argument is not used in the event, we write $(id, m, \perp, d_o)$ (resp. $(id, m, d_i, \perp)$). We will assume that each event has a unique event identifier. We will use $E_{\mathcal{D}}$ to denote the set of all events of $\mathcal{D}$. A duplicate-free sequence over $E_{\mathcal{D}}$ is called a $\mathcal{D}$-behavior. The *semantics* of data structure $\mathcal{D}$ is a set of $\mathcal{D}$-behaviors, each of which is called a *valid* behavior. For each data structure $\mathcal{D}$, we will define a labelled transition system $\mathsf{LTS}_{\mathcal{D}}$ such that $\mathbf{e} \in \mathsf{Tr}(\mathsf{LTS}_{\mathcal{D}})$ iff $\mathbf{e}$ is a valid $\mathcal{D}$-behavior. Below we list the data structures that we will consider in this paper.

2.1.1 Pool, $\mathcal{P}$

The method alphabet $\Sigma_{\mathcal{P}}$ of a pool is the set $\{\texttt{put}, \texttt{take}\}$. Events of $\mathcal{P}$ are written as $\texttt{put}^{id}(x)$, short for $(id, \texttt{put}, x, \perp)$, and $\texttt{take}^{id}(x)$, short for $(id, \texttt{take}, \perp, x)$. For conciseness, from this point on we will omit the superscript id. Events with `put` are called *put* events,

and those with `take` are called *take* events. We use `Put` and `Take` to denote the set of of all put and take events, respectively.

$\mathsf{LTS}_{\mathcal{P}}$ is defined as $(Pow(\mathbb{N}), \emptyset, E_{\mathcal{P}}, \rightarrow_{\mathcal{P}})$, where $\rightarrow_{\mathcal{P}}$ is defined as:

- $q \xrightarrow{\texttt{put}(x)}_{\mathcal{P}} q'$ iff $q' = q \cup \{x\}$,
- $q \xrightarrow{\texttt{take}(x)}_{\mathcal{P}} q'$ iff $x \in q$ and $q' = q \setminus \{x\}$,
- $q \xrightarrow{\texttt{take}(\texttt{NULL})}_{\mathcal{P}} q'$ iff $q = q' = \emptyset$.

2.1.2 Pool with Membership, $\mathcal{P}^?$

The method alphabet $\Sigma_{\mathcal{P}^?}$ is $\Sigma_{\mathcal{P}} \cup \{\texttt{mem}\}$. An event of $\mathcal{P}^?$ is either a $\mathcal{P}$ event or of the form $\texttt{mem}(x, y)$, short for $(id, \texttt{mem}, x, y)$. Events with `mem` are called *query* events, and `Mem` denotes the set of all query events.

$\mathsf{LTS}_{\mathcal{P}^?}$ is defined as $(Pow(\mathbb{N}), \emptyset, E_{\mathcal{P}^?}, \rightarrow_{\mathcal{P}^?})$, where $\rightarrow_{\mathcal{P}^?}$ is defined as:

- $q \xrightarrow{x}_{\mathcal{P}^?} q'$ if $q \xrightarrow{x}_{\mathcal{P}} q'$,
- $q \xrightarrow{\texttt{mem}(x,y)}_{\mathcal{P}^?} q'$ iff $q = q'$, and either $y = x$ and $x \in q$, or $y = x + 1$ and $x \notin q$.

2.1.3 Queue, $\mathcal{Q}$

The method alphabet $\Sigma_{\mathcal{Q}}$ is the set $\{\texttt{enq}, \texttt{deq}\}$. Events of $\mathcal{Q}$ are written as $\texttt{enq}(x)$, short for $(id, \texttt{enq}, x, \perp)$, and $\texttt{deq}(x)$, short for $(id, \texttt{deq}, \perp, x)$. Events with `enq` are called *enqueue* events, and those with `deq` are called *dequeue* events. We use `Enq` and `Deq` to denote the set of all enqueue and dequeue events, respectively.

$\mathsf{LTS}_{\mathcal{Q}}$ is defined as $(\mathbb{N}^*, \varepsilon, E_{\mathcal{Q}}, \rightarrow_{\mathcal{Q}})$, where $\rightarrow_{\mathcal{Q}}$ is defined as:

- $q \xrightarrow{\texttt{enq}(x)}_{\mathcal{Q}} q'$ iff $q' = q \cdot x$,
- $q \xrightarrow{\texttt{deq}(x)}_{\mathcal{Q}} q'$ and $x \neq \texttt{NULL}$ iff $q = x \cdot q'$,
- $q \xrightarrow{\texttt{deq}(\texttt{NULL})}_{\mathcal{Q}} q'$ iff $q = q' = \varepsilon$.

2.1.4 Stack, $\mathcal{S}$

The method alphabet $\Sigma_{\mathcal{S}}$ is the set $\{\texttt{push}, \texttt{pop}\}$. Events of $\mathcal{S}$ are written as $\texttt{push}(x)$, short for $(id, \texttt{push}, x, \perp)$, and $\texttt{pop}(x)$, short for $(id, \texttt{pop}, \perp, x)$. Events with `push` are called *push* events, and those with `pop` are called *pop* events. We use `Push` and `Pop` to denote the set of all push and pop events, respectively.

$\mathsf{LTS}_{\mathcal{S}}$ is defined as $(\mathbb{N}^*, \varepsilon, E_{\mathcal{S}}, \rightarrow_{\mathcal{S}})$, where $\rightarrow_{\mathcal{S}}$ is defined as:

- $q \xrightarrow{\texttt{push}(x)}_{\mathcal{S}} q'$ iff $q' = q \cdot x$,
- $q \xrightarrow{\texttt{pop}(x)}_{\mathcal{S}} q'$ and $x \neq \texttt{NULL}$ iff $q = q' \cdot x$,
- $q \xrightarrow{\texttt{pop}(\texttt{NULL})}_{\mathcal{S}} q'$ iff $q = q' = \varepsilon$.

2.1.5 Register, $\mathcal{R}$

The method alphabet $\Sigma_{\mathcal{R}}$ is the set $\{\texttt{wr}_i, \texttt{rd}_i \mid i \in \mathbb{N}\}$. Events of $\mathcal{R}$ are written as $\texttt{wr}_i(x)$, short for $(id, \texttt{wr}_i, x, \perp)$, and $\texttt{rd}_i(x)$, short for $(id, \texttt{rd}_i, \perp, x)$. Events with $\texttt{wr}_i$ are called *write* events, and those with $\texttt{rd}_i$ are called *read* events. We use `Wr` and `Rd` to denote the set of all write and read events, respectively.

$\mathsf{LTS}_{\mathcal{R}}$ is defined as $(\mathbb{N}[\mathbb{N}], \lambda x.0, E_{\mathcal{R}}, \rightarrow_{\mathcal{R}})$, where $\rightarrow_{\mathcal{R}}$ is defined as:

- $q \xrightarrow{\texttt{wr}(i,x)}_{\mathcal{R}} q'$ iff $q' = q[i \mapsto x]$.
- $q \xrightarrow{\texttt{rd}(i,x)}_{\mathcal{R}} q'$ iff $q' = q$ and $q(i) = x$.

2.2 Histories

Each event $e = (uid, m, d_i, d_o)$ generates two *actions*: the *invocation* of e, written as $inv(e)$, and the *response* of e, written as $res(e)$. We will also use $m_i^{uid}(d_i)$ and $m_r^{uid}(d_o)$ to denote the invocation and response actions, respectively. When a particular method m does not have an input (resp. output) parameter, we will write m_i^{uid} (resp. m_r^{uid}) for the corresponding invocation (resp. response) action. We will also often omit the superscripts, when they are not important. For an event set E, let E_i and E_r denote the set of all invocation and response actions generated by E.

A $\mathcal{D}$-*history* is a sequence of invocation and response actions generated by $E_{\mathcal{D}}$. The unique identifier of each event unambigiously pairs each invocation action to a unique response action; in such a case, the actions are said to *match*. We will make use of this pairing without explicitly referring to event identifiers when there is no confusion. Similarly, we will omit $\mathcal{D}$ whenever it is either inconsequential or clear from the text. A history $\mathbf{h}$ is *well-formed* if every response action appears after its matching invocation action in $\mathbf{h}$. An event e is *completed* in $\mathbf{h}$, if both of its invocation and response actions appear in $\mathbf{h}$. Formally, e is completed if $e_i, e_r \in Set(\mathbf{h})$. A history $\mathbf{h}$ is *complete* if for all events e, $e_i \in Set(\mathbf{h})$ iff $e_r \in Set(\mathbf{h})$. In what follows we will consider only well-formed and complete histories.

An event e precedes another event e' in $\mathbf{h}$, written $e \prec_{\mathbf{h}} e'$, if the response action of e appear before the invocation action of e'; i.e. $e_r <_{\mathbf{h}} e'_i$. A history is called *sequential* if all invocation actions are immediately followed by their matching responses. Formally, the (complete) history $\mathbf{h}$ is sequential if it is of the form $e_{1,i} \cdot e_{1,r} \cdot \ldots \cdot e_{n,i} \cdot e_{n,r}$. We identify sequential $\mathcal{D}$-histories with $\mathcal{D}$-behaviors by mapping each matching pair of invocation and response actions to the event generating them. A sequential history $\mathbf{s}$ is a *linearization* of a history $\mathbf{h}$, if $\mathbf{s}$ is a permutation of $\mathbf{h}$ such that $e \prec_{\mathbf{h}} e'$ implies $e \prec_{\mathbf{s}} e'$.

▶ **Definition 1** (Linearizability). A $\mathcal{D}$-history $\mathbf{h}$ is linearizable if there exists a linearization of $\mathbf{h}$ that is a legal $\mathcal{D}$-behavior. A set H of histories is linearizable if every $\mathbf{h} \in H$ is linearizable.

Let T be the set of *thread id*'s. A threaded $\mathcal{D}$-action is of the form (t, e), where $t \in T$ and $e \in E_{\mathcal{D},i} \cup E_{\mathcal{D},r}$. For a threaded action $\vec{a} = (t, a)$, we use tid and act to retrieve the first and second components, respectively; i.e. $tid(\vec{a}) = t$ and $act(\vec{a}) = a$. Similarly, tid and act are point-wise extended over sequences of threaded-actions. For a sequence of threaded-actions $\vec{\mathbf{h}}$ and a thread id $t \in T$, let $\vec{\mathbf{h}} \downarrow_t$ denote the subsequence obtained by removing all threaded-actions from $\vec{\mathbf{h}}$ whose first component is not equal to t. A *threaded* $\mathcal{D}$-history is a sequence $\vec{\mathbf{h}}$ over threaded actions such that

- the sequence $act(\vec{\mathbf{h}})$ is a complete $\mathcal{D}$-history,
- the sequence $act(\vec{\mathbf{h}} \downarrow_t)$ is sequential for any $t \in T$.

The second condition implies that at any point in a threaded history any thread $t \in T$ can have at most one unmatched action. For ease of presentation, we will use $\mathbf{h}$ and $\mathbf{h}(t)$ to denote $act(\vec{\mathbf{h}})$ and $act(\vec{\mathbf{h}} \downarrow_t)$, respectively. We will extend the properties of histories to threaded histories: a threaded history $\vec{\mathbf{h}}$ is said to satisfy property P if $\mathbf{h}$ satisfies P.

▶ **Definition 2** (Sequential Consistency). A threaded $\mathcal{D}$-history is sequentially consistent if there is a sequential threaded $\mathcal{D}$-history $\vec{\mathbf{s}}$ such that $\vec{\mathbf{s}}$ is a permutation of $\vec{\mathbf{h}}$ and for all $t \in T$ we have $\mathbf{s}(t) = \mathbf{h}(t)$.

Intuitively, for a threaded history $\vec{\mathbf{h}}$ to be sequentially consistent (SC) only the relative ordering per thread has to be respected. Since by the definition of threaded-histories, if (t, e) and (t, e') are both in $\vec{\mathbf{h}}$, then either $e \prec_{\mathbf{h}} e'$ or $e' \prec_{\mathbf{h}} e$, linearizability must preserve the

relative ordering among events done by the same thread. In other words, linearizability is a stronger condition than sequential consistency.

▶ Fact 3. Let $\vec{\mathbf{h}}$ be a threaded $\mathcal{D}$-history. It is sequentially consistent if it is linearizable.

As far as the implementation of data structures is concerned, we will not specify a particular programming language. We will assume that each method in the method alphabet has an accompanying procedure. An *execution trace* is a sequence of instruction labels coupled with thread identifiers executing the instruction. For instance, $(t : i)$ denotes the execution of instruction with the unique label i by thread t. An instruction label is the *entry* point of method m, written $enter(m)$, if it is the label of the first instruction of m. Similarly, an instruction label is an *exit* point of m, written $exit(m)$, if it is the lable of an instruction that completes the execution of m. Each execution trace τ induces a history $h(\tau)$ which is obtained by replacing each $(t : enter(m))$ with $m_i^{t_{uid}}(d_i)$, each $(t : exit(m))$ with $m_r^{t_{uid}}(d_o)$, and removing the remaining (intermediate) symbols. We assume that states of an execution trace contain enough information to deduce the values of d_i and d_o associated with each entry and exit point. An execution trace is *complete* if its induced history is complete. For an execution trace τ and some $t \in T$, let $\tau \downarrow_t$ denote the execution trace obtained by retaining only the symbols due to t (symbols of the form $(t : i)$. An implementation is identified with the set of execution traces it generates. When clear from the context we will refer to the induced history of an execution trace as a history of the implementation.

3 Properties of Data Structures

Our main result is that sequential consistency is too weak because the class of SC implementations includes *bad* ones. In order to generalize our result, we abstract away irrelevant specifics of data structures and extract what seems to be the essential property that causes SC implementations to misbehave. We identify two properties: composability and robustness. A data structure is composable if any two valid behavior can be interleaved in such a way that the result is also a valid behavior. Robustness means that the data structure has events that are state independent. In this section, we formalize these notions.

▶ **Definition 4** (Composable)**.** Let $\mathbf{e}$ and $\mathbf{f}$ be two valid $\mathcal{D}$-behaviors. They are called *composable* if there exist two partitionings $\mathbf{e} = \mathbf{e}_1 \dots \mathbf{e}_k$ and $\mathbf{f} = \mathbf{f}_1 \dots \mathbf{f}_k$ such that the behavior $\mathbf{e}_1\mathbf{f}_1 \dots \mathbf{e}_k\mathbf{f}_k$ is a valid $\mathcal{D}$-behavior. The data structure $\mathcal{D}$ is called *composable* if any pair of valid $\mathcal{D}$-behaviors are composable.

Informally, two valid $\mathcal{D}$-behaviors are composable if it is possible to interleave them to obtain another valid $\mathcal{D}$-behavior. We now proceed with a series of results, establishing composability of the data structures we consider in this paper.

▶ **Lemma 5** (Pool and Composability)**.** *The pool data structure $\mathcal{P}$ is composable.*

Proof. Let $\mathbf{e}$ and $\mathbf{f}$ be two valid $\mathcal{P}$-sequences. Let $\mathbf{e}_1$ and $\mathbf{f}_1$ be the maximal prefixes of $\mathbf{e}$ and $\mathbf{f}$, respectively, such that $last(\mathbf{e}_1) = \texttt{take}(\texttt{NULL})$ and $last(\mathbf{f}_1) = \texttt{take}(\texttt{NULL})$. Let $\mathbf{e}_2$ and $\mathbf{f}_2$ be the remaining suffixes of $\mathbf{e}$ and $\mathbf{f}$. That is, $\mathbf{e} = \mathbf{e}_1\mathbf{e}_2$ and $\mathbf{f} = \mathbf{f}_1\mathbf{f}_2$. We claim that $\mathbf{g} = \mathbf{e}_1\mathbf{f}_1\mathbf{e}_2\mathbf{f}_2$ is a valid $\mathcal{P}$-behavior. This is equivalent to showing that there is a run of $\mathsf{LTS}_{\mathcal{P}}$ whose trace is $\mathbf{g}$. By the validity of $\mathbf{e}$, we know that there is a run $\mathbf{r}_1$ of $\mathsf{LTS}_{\mathcal{P}}$ with trace $\mathbf{e}_1$ because $\mathsf{Tr}(\mathsf{LTS}_{\mathcal{P}})$ is prefix-closed. Furthermore, by the assumption that $last(\mathbf{e}_1) = \texttt{take}(\texttt{NULL})$, that run ends at the state $\emptyset$. Similarly, there is a run $\mathbf{r}_2$ with trace $\mathbf{f}_1$. Since $\mathbf{r}_1$ ends at $\emptyset$, $\mathbf{r}_1\mathbf{r}_2$ is also a run of $\mathsf{LTS}_{\mathcal{P}}$ whose trace is $\mathbf{e}_1\mathbf{f}_1$. Since the trace $\mathbf{e}_2$ belongs to a run $\mathbf{r}_3$ (implying that $\mathbf{r}_1\mathbf{r}_3$ is the run with trace $\mathbf{e}$) which starts at $\emptyset$, $\mathbf{r}_1\mathbf{r}_2\mathbf{r}_3$ is a run of $\mathsf{LTS}_{\mathcal{P}}$. Finally, we have to

extend $\mathbf{r}_1\mathbf{r}_2\mathbf{r}_3$ with a run $\mathbf{r}_4$ whose trace is $\mathbf{f}_2$. We proceed by induction on the length of $\mathbf{f}_2$. The base case, $\mathbf{f}_2 = \varepsilon$ is trivial. Assume that the next event is $\texttt{put}(x)$. By the definition of $\mathsf{LTS}_{\mathcal{P}}$, such a transition is enabled at all states. Assume that the next event is $\texttt{take}(x)$ (note that by construction, $\mathbf{f}_2$ does not contain a transition with label $\texttt{take}(\texttt{NULL})$). Because the run with trace $\mathbf{f}_2$ starts at the state $\emptyset$, there must have been an event $\texttt{put}(x)$ in $\mathbf{f}_2$ and that no event since the last occurrence in $\mathbf{f}_2$ prior to $\texttt{take}(x)$ was equal to $\texttt{take}(x)$. In other words, we must have $x \in q$, where q is the current state, which means that it is possible to extend the run with $\texttt{take}(x)$. ◀

▶ **Lemma 6** (Stack and Composability). *The stack data structure $\mathcal{S}$ is composable.*

Proof (Sketch). The construction is similar to the one given in the proof of $\mathcal{P}$. For any two valid $\mathcal{S}$-behaviors $\mathbf{e}$ and $\mathbf{f}$, we take the maximal prefixes $\mathbf{e}_1$ and $\mathbf{f}_1$ both of which end with $\texttt{pop}(\texttt{NULL})$. Then, the composed behavior $\mathbf{e}_1\mathbf{f}_1\mathbf{e}_2\mathbf{f}_2$ is also a valid $\mathcal{S}$-behavior, where $\mathbf{e}_2$ and $\mathbf{f}_2$ are the remaining suffixes of $\mathbf{e}$ and $\mathbf{f}$. Intuitively, the constructed behavior is valid because neither $\mathbf{e}_2$ nor $\mathbf{f}_2$ reaches beyond what they have pushed onto the stack (no appearance of $\texttt{pop}(\texttt{NULL})$ in either of the two). ◀

▶ **Lemma 7** (Queue and Composability). *The queue data structure $\mathcal{Q}$ is composable.*

Proof (Sketch). Unlike the previous two cases, the construction for queue is more involved. We will need to partition a valid $\mathcal{Q}$-behavior according to the relative ordering between the enqueue events of $\texttt{Enq}$ and the dequeue events of $\texttt{Deq}$. For any $\texttt{enq}(x)$, let $\texttt{deq}(x)$ be called its *observer*. Given a valid $\mathcal{Q}$-behavior $\mathbf{e}$, we define $Era_j(\mathbf{e})$ inductively as follows:

- $Era_0(\mathbf{e}) = \varepsilon$.
- $Era_{j+1}(\mathbf{e})$ the maximum segment that begins with an enqueue event whose observer is in $Era_j(\mathbf{e})$ and extends until the first element that belongs to $Era_j(\mathbf{e})$.

To illustrate the definition, consider the valid $\mathcal{Q}$-behavior:

$$\mathbf{e} \quad = \quad \texttt{enq}(1) \cdot \texttt{enq}(2) \cdot \texttt{deq}(1) \cdot \texttt{enq}(3) \cdot \texttt{deq}(2) \cdot \texttt{enq}(4) \cdot \texttt{enq}(5) \cdot \texttt{deq}(3) \cdot \texttt{deq}(4)$$

Then, we have

$$\begin{aligned} Era_1(\mathbf{e}) &= \texttt{enq}(5) \cdot \texttt{deq}(3) \cdot \texttt{deq}(4) \\ Era_2(\mathbf{e}) &= \texttt{enq}(3) \cdot \texttt{deq}(2) \cdot \texttt{enq}(4) \\ Era_3(\mathbf{e}) &= \texttt{enq}(2) \cdot \texttt{deq}(1) \\ Era_4(\mathbf{e}) &= \texttt{enq}(1) \end{aligned}$$

Observe that by construction all enqueue events of $Era_j(\mathbf{e})$ for $j > 1$ are observed by the dequeue events of $Era_{j-1}(\mathbf{e})$. By convention, for any $n > k$ such that $Era_k(\mathbf{e}) \ldots Era_1(\mathbf{e}) = \mathbf{e}$, we set $Era_n(\mathbf{e}) = \varepsilon$.

Now let $\mathbf{e}$ and $\mathbf{f}$ be two valid $\mathcal{Q}$-behaviors. Let $\mathbf{e}_1$ be the maximal prefix of $\mathbf{e}$ such that the run with trace $\mathbf{e}_1$, which necessarily exists, ends at state ε (the initial state of $\mathsf{LTS}_{\mathcal{Q}}$). Let $\mathbf{e}_r$ be the remaining suffix of $\mathbf{e}$; i.e. $\mathbf{e} = \mathbf{e}_1\mathbf{e}_r$. Let $\mathbf{f}_1$ and $\mathbf{f}_r$ be defined similarly for $\mathbf{f}$. Now construct the Era sequences for $\mathbf{e}_r$ and $\mathbf{f}_r$. Let j_e be the maximal index of a non-empty era sequence of $\mathbf{e}$, j_f be the index for $\mathbf{f}$. Without loss of generality, assume that $j_e \geq j_f$. Then, the interleaving

$$\mathbf{e}_1 \cdot \mathbf{f}_1 \cdot Era_{e_j}(\mathbf{e}) \cdot Era_{e_j}(\mathbf{f}) \ldots Era_1(\mathbf{e}) \cdot Era_1(\mathbf{f})$$

is a valid $\mathcal{Q}$-behavior. To illustrate the construction, consider another valid $\mathcal{Q}$-behavior

$$\mathbf{f} \quad = \quad \texttt{enq}(6) \cdot \texttt{deq}(6) \cdot \texttt{enq}(7) \cdot \texttt{enq}(8) \cdot \texttt{deq}(7) \cdot \texttt{enq}(9) \cdot \texttt{enq}(10) \cdot \texttt{deq}(8)$$

Then, $\mathbf{f}_1$ is $\texttt{enq}(6) \cdot \texttt{deq}(6)$ and the era sequences for the suffix $\mathbf{f}_r$ is

$$\begin{aligned} Era_1(\mathbf{f}) &= \texttt{enq}(9) \cdot \texttt{enq}(10) \cdot \texttt{deq}(8) \\ Era_2(\mathbf{f}) &= \texttt{enq}(8) \cdot \texttt{deq}(7) \\ Era_3(\mathbf{f}) &= \texttt{enq}(7) \end{aligned}$$

Finally, the interleaving for $\mathbf{e}$ and $\mathbf{f}$ is given as:

$$\begin{aligned} &\texttt{enq}(6) \cdot \texttt{deq}(6) \cdot \texttt{enq}(1) \cdot \texttt{enq}(2) \cdot \texttt{deq}(1) \cdot \texttt{enq}(7) \cdot \texttt{enq}(3) \cdot \texttt{deq}(2) \cdot \\ &\qquad \texttt{enq}(4) \cdot \texttt{enq}(8) \cdot \texttt{deq}(7) \cdot \texttt{enq}(5) \cdot \texttt{deq}(3) \cdot \texttt{deq}(4) \cdot \texttt{enq}(9) \cdot \texttt{enq}(10) \cdot \texttt{deq}(8) \end{aligned}$$

which is a valid $\mathcal{Q}$-behavior. Intuitively, the construction works as each era sequence of one behavior (say $\mathbf{e}$) with index j removes all the elements inserted by the most recent era sequence of the same behavior with index $j+1$, thereby making the insertions of this behavior ($\mathbf{e}$) invisible to the other behavior ($\mathbf{f}$) and vice versa. ◀

The other two remaining data structures, $\mathcal{P}^?$ and $\mathcal{R}$, are not composable. For the pool with membership data structure $\mathcal{P}^?$, the following two valid $\mathcal{P}^?$-behaviors have no interleaving that is a valid $\mathcal{P}^?$-behavior:

$$\mathbf{e} = \texttt{put}(1) \cdot \texttt{mem}(2,3), \quad \mathbf{f} = \texttt{put}(2) \cdot \texttt{mem}(1,2)$$

since $\texttt{mem}(2,3)$ has to come before $\texttt{put}(2)$ and $\texttt{mem}(1,2)$ has to come before $\texttt{put}(1)$, both conditions of which cannot be simultaneously satisfied.

Similarly, for the register data structure $\mathcal{R}$, the following valid $\mathcal{R}$-behaviors are not composable:

$$\mathbf{e} = \texttt{wr}(1,1) \cdot \texttt{rd}(2,0), \quad \mathbf{f} = \texttt{wr}(2,1) \cdot \texttt{rd}(1,0)$$

The next property, robustness, is satisfied by all five of the data structures we consider and to the best of our knowledge all data structures are robust.

▶ **Definition 8** (Robustness)**.** Let $\mathbf{e}$ be a valid $\mathcal{D}$-behavior. It is *robust* if for any state $q \in \mathsf{LTS}_{\mathcal{D}}$, there is a run that starts at q with trace $\mathbf{e}$. A data structure is robust if it contains at least one robust sequence.

In general, any data structure which contains a *total* event (e.g. $\texttt{enq}(x)$ of $\mathcal{Q}$ or $\texttt{push}(x)$ of $\mathcal{S}$) is robust.

▶ **Lemma 9.** *All of $\mathcal{P}$, $\mathcal{P}^?$, $\mathcal{Q}$, $\mathcal{S}$, $\mathcal{R}$ are robust.*

Proof. The event $\texttt{take}(x)$ is enabled at every state of $\mathsf{LTS}_{\mathcal{P}}$ and $\mathsf{LTS}_{\mathcal{P}^?}$. The event $\texttt{enq}(x)$ is enabled at every state of $\mathsf{LTS}_{\mathcal{Q}}$. The event $\texttt{push}(x)$ is enabled at every state of $\mathsf{LTS}_{\mathcal{S}}$. The event $\texttt{wr}(x,y)$ is enabled at every state of $\mathsf{LTS}_{\mathcal{R}}$. ◀

Robust sequences can contain arbitrary events or be restricted to a subset of events, depending on the data structure.

▶ **Lemma 10.** *A robust sequence of $\mathcal{P}$ and $\mathcal{P}^?$ cannot contain* $\texttt{put}(\texttt{NULL})$*; for $\mathcal{Q}$, it cannot contain* $\texttt{deq}(\texttt{NULL})$*; for $\mathcal{S}$, it cannot contain* $\texttt{pop}(\texttt{NULL})$*. There is no restriction on robust sequences for $\mathcal{R}$.*

4 SC is too Weak

In this section, we present the bad implementations that SC seems to allow. These bad implementations come in two variants: conditional and unconditional non-synchronization. We show that all robust data structures allow conditional non-sychronization. Unconditional non-synchronization, arguably the worst of the two, is allowed by composable data structures $(\mathcal{P}, \mathcal{Q}, \mathcal{S})$.

Let us call a label l *enabled* at state q if there exists a state q' such that $q \xrightarrow{l} q'$.

▶ **Definition 11** (Initialized)**.** Let $(Q, q_0, L, \rightarrow)$ be an LTS. A state $q \in Q$ is *subsumed* by another state $q' \in Q$ if for all $l \in L$, l is enabled at q implies l is enabled at q'. The LTS is called *initialized* if its initial state q_0 is not subsumed by any other state.

We call an LTS $(Q, q_0, L, \rightarrow)$ *non-trivial* if there is at least one state $q \neq q_0$ and one label $l \in L$ such that $q_0 \xrightarrow{l} q$. A data structure $\mathcal{D}$ is non-trivial if $\mathsf{LTS}_{\mathcal{D}}$ is non-trivial.

▶ **Lemma 12.** *The LTS corresponding to $\mathcal{P}, \mathcal{P}^?, \mathcal{Q}, \mathcal{S}, \mathcal{R}$ are initialized.*

Proof. In all cases, there are transitions which are only enabled at the initial state. The transitions for all except for $\mathsf{LTS}_{\mathcal{R}}$ are $\texttt{take}(\texttt{NULL})$ in $\mathsf{LTS}_{\mathcal{P}}$ and $\mathsf{LTS}_{\mathcal{P}^?}$; $\texttt{deq}(\texttt{NULL})$ in $\mathsf{LTS}_{\mathcal{Q}}$; $\texttt{pop}(\texttt{NULL})$ in $\mathsf{LTS}_{\mathcal{S}}$. For $\mathsf{LTS}_{\mathcal{R}}$, let $q' \neq q_0$ be some state. By definition, there must be at least one $x \in \mathbb{N}$ such that $q'(x) \neq 0$ since otherwise q' and $q_0 = \lambda x.0$ are identical. Then, $\texttt{rd}(x, 0)$ is enabled at q_0 but not at q'. ◀

For each data structure $\mathcal{D}$, we distinguish an implementation $\mathsf{Imp}_{iso}(\mathcal{D})$, called the *isolated implementation* of $\mathcal{D}$, whose induced histories are thread-locally valid $\mathcal{D}$-behaviors. Formally, for any execution trace τ of $\mathsf{Imp}_{iso}(\mathcal{D})$ and for any $t \in Tid$, the induced history of $\tau \downarrow_t$ is a valid $\mathcal{D}$-behavior. Intuitively, isolated implementations are those which do not need any communication between threads. For most data structures, such implementations are not desirable and as the following result shows are ruled out by linearizability.

▶ **Lemma 13.** *If $\mathcal{D}$ is initialized and non-trivial, then $\mathsf{Imp}_{iso}(\mathcal{D})$ is not linearizable.*

Proof. Let $\mathsf{LTS}_{\mathcal{D}}$ be the tuple $(Q, q_0, E, \rightarrow)$. Because $\mathcal{D}$ is non-trivial, there is a transition $q_0 \xrightarrow{e} q$ for some $e = (m, d_i, d_o) \in E$ and $q \neq q_0$. Because $\mathcal{D}$ is initialized, there is an event $e' \in E$ such that $e' = (m', {d'}_i, {d'}_o)$ is enabled at q_0 and not at q. Then consider the history $\mathbf{h}$ for $t, t' \in T$:

$$\mathbf{h} \stackrel{def}{=} (t, m_i(d_i)) \cdot (t, m_r(d_o)) \cdot (t', m'_i({d'}_i)) \cdot (t', m'_r({d'}_o))$$

The only linearization for this history is $(m, d_i, d_o) \cdot (m', d'_i, d'_o)$. Since this is not a valid $\mathcal{D}$-behavior, $\mathbf{h}$ is not linearizable. However, because both (m, d_i, d_o) and (m', d'_i, d'_o) are individually valid $\mathcal{D}$-behaviors, $\mathbf{h}$ is induced by some execution trace of $\mathsf{Imp}_{iso}(\mathcal{D})$, implying that the latter is not linearizable. ◀

The following result immediately follows from Lemma's 12 and 13.

▶ **Corollary 14.** *The isolated implementations of $\mathcal{P}$, $\mathcal{P}^?$, $\mathcal{Q}$, $\mathcal{S}$ and $\mathcal{R}$ are not linearizable.*

The previous result shows that the definition of linearizability is strong enough to leave out these pathological implementations. As we show next, sequential consistency is weak enough to allow for isolated implementations of some data structures.

▶ **Theorem 15** (SC and Isolated Implementations). *If $\mathcal{D}$ is composable, then $\mathsf{Imp}_{iso}(\mathcal{D})$ is sequentially consistent.*

Proof. Let τ be an execution trace of $\mathsf{Imp}_{iso}(\mathcal{D})$ and let $\vec{\mathbf{h}}$ be the induced threaded-history. We do induction on the number of threads that execute at least one method during τ. If $\vec{\mathbf{h}}$ is due to a single thread, then it is a valid $\mathcal{D}$-behavior and hence SC. Assume that if $\vec{\mathbf{h}}$ has less than or equal to k different threads, it is SC. Now let $\vec{\mathbf{h}}$ have $k+1$ different threads and let $t \neq t'$ be the identifiers of two of those. By the definition of $\mathsf{Imp}_{iso}(\mathcal{D})$, both $\mathbf{h}(t)$ and $\mathbf{h}(t')$ are valid $\mathcal{D}$-behaviors. By composability, there is an interleaving $\mathbf{h}'$ of $\mathbf{h}(t)$ and $\mathbf{h}(t')$ which is also a valid $\mathcal{D}$-behavior. Let $u \in T$ be a thread identifier that does not appear in $\vec{\mathbf{h}}$ and let $\vec{\mathbf{h}}'$ denote the threaded-history obtained by coupling each symbol of $\mathbf{h}'$ with u. That is, $\vec{\mathbf{h}}'$ is the sequence $\langle u, \mathbf{h}'(i)\rangle_{i\in[1,len(\mathbf{h}')]}$. Let $\vec{\mathbf{g}}$ be the threaded history which is constructed by first projecting out from $\vec{\mathbf{h}}$ all symbols s with $act(s) = t$ or $act(s) = t'$, and then extending it with $\vec{\mathbf{h}}'$. By construction, $\vec{\mathbf{g}}$ has k different threads and each $\mathbf{g}(t)$ is a valid $\mathcal{D}$-behavior. By definition, there is an execution trace τ' which induces $\vec{\mathbf{g}}$. By inductive hypothesis, there is a sequential threaded valid $\mathcal{D}$-history $\vec{\mathbf{s}}$ such that for all $t \in T$ we have $\vec{\mathbf{s}}(t) = \vec{\mathbf{g}}(t)$. Finally, because $\vec{\mathbf{h}}'$ was an interleaving of $\vec{\mathbf{h}}(t)$ and $\vec{\mathbf{h}}(t')$, replacing u in $\vec{\mathbf{s}}$ with the original t or t' identifiers yields the valid sequential threaded $\mathcal{D}$-history corresponding to $\vec{\mathbf{h}}$. ◀

This implies that the definition of sequential consistency is not strong enough for composable data structures.

▶ **Corollary 16.** *The isolated implementations of $\mathcal{P}$, $\mathcal{Q}$ and $\mathcal{S}$ are sequentially consistent.*

An execution trace τ of a $\mathcal{D}$ implementation is called *t-singular* if $h(\tau)\uparrow^t$ is linearizable and $h(\tau)\downarrow_t$ is a valid $\mathcal{D}$-behavior. Let $\mathsf{Imp}_{sing}(\mathcal{D})$, the *singular implementation* of $\mathcal{D}$, denote the union of all linearizable execution traces and all t-singular execution traces. We next show that singular implementations of robust data structures are non-linearizable.

▶ **Lemma 17.** *If $\mathcal{D}$ is robust, initialized and non-trivial, then $\mathsf{Imp}_{sing}(\mathcal{D})$ is not linearizable.*

Proof. Let $\mathsf{LTS}_{\mathcal{D}}$ be the tuple $(Q, q_0, E, \rightarrow)$. Let $\mathbf{e} = \langle e(i)\rangle_{i\in[1,n]}$ be a robust sequence. Let $q_0 \xrightarrow{e_1} q_1 \ldots \xrightarrow{e_n} q_n$ be the run whose trace is $\mathbf{e}$. If $q_n \neq q_0$, because $\mathcal{D}$ is initialized, there must be some event $e' \in E$ enabled at q_0 but not at q_n. Then, similar to the proof of Lemma 13, the threaded history in which thread $t \in T$ runs all actions associated with $\mathbf{h}$ followed by another thread $u \in T$ running the two actions generated by the event e is in $\mathsf{Imp}_{sing}(\mathcal{D})$ because e represents a valid $\mathcal{D}$-behavior and $\mathbf{e}$ represents a robust sequence. If $q_n = q_0$, then there must be a state $q' \neq q_0$ and a label e' such that $q_0 \xrightarrow{e'} q'$ holds. The rest of the argument is the same as the previous one. ◀

The following is immediate from Lemma 9.

▶ **Corollary 18.** *The singular implementations of $\mathcal{P}$, $\mathcal{P}^?$, $\mathcal{Q}$, $\mathcal{S}$ and $\mathcal{R}$ are not linearizable.*

Intuitively, in singular implementations the behavior of some thread t can be hidden from the rest of the system as long as the sequence generated by t remains robust. This, although arguably not as bad as being isolated, is still an undesirable feature and linearizability forbids it. We now show that singular implementations are sequentially consistent.

▶ **Theorem 19** (SC and Singular Implementations). *For any data structure $\mathcal{D}$, $\mathsf{Imp}_{sing}(\mathcal{D})$ is sequentially consistent.*

Proof. Let τ be an execution trace of $\mathsf{Imp}_{sing}(\mathcal{D})$ and let $\vec{\mathbf{h}}$ be the induced threaded history. If $\vec{\mathbf{h}}$ is linearizable, then by Fact. 3 it is also sequentially consistent. If $\vec{\mathbf{h}}$ is not linearizable, then there must be some thread $t \in T$ such that $h(\tau)\downarrow_t$ is robust. Furthermore, we know that $h(\tau)\uparrow^t$ is linearizable. Assume that $\mathbf{s}$ is the linearization of $h(\tau)\uparrow^t$. Then the sequence $\mathbf{s}'$ formed by appending $h(\tau)\downarrow_t$ to $\mathbf{s}$ is a valid $\mathcal{D}$-behavior. Since for all $u \in T$ we have $\mathbf{s}'\downarrow_u = h(\tau)\downarrow_u$, we conclude that $\vec{\mathbf{h}}$ is sequentially consistent. ◀

We end this section by giving templates for isolated and singular implementations. Assume that we already have a sequential implementation of any data structure and use the notation $\texttt{Obj}_{\mathcal{D}}$ to denote the class implementing the methods of $\mathcal{D}$. For instance, for the $\mathcal{Q}$ data structure, $\texttt{Obj}_{\mathcal{Q}}$ implements the required methods which are called by appending the method to an object O of type $\texttt{Obj}_{\mathcal{Q}}$ as in $O.\texttt{enq}(x)$. In our programs, we assume that each thread $t \in T$ has its thread-local copy of type $\texttt{Obj}_{\mathcal{D}}$ and use the notation $\texttt{Obj}[t]$ to denote the object exclusively used by t. Then, in an isolated implementation, each method $m \in \Sigma_{\mathcal{D}}$ has the following template: Here self evaluates to t whenever the method is run by thread t.

$m(d_i)$ { d_o = $\texttt{Obj}$[self].$m(d_i)$; **return** d_o; }

Event $d_o = m(d_i)$

```
if self=i then
    d_o ← Obj[self].m(d_i);
    newseq ← lseq·m(d_i,d_o);
    if notRobust(lseq) then
        atomic <commit(lseq)>;
        atomic <d_o ← O.m(d_i)>;
        lseq ← ε;
    else
        lseq ← newseq;
else
    atomic <d_o ← O.m(d_i)>;
return d_o;
```

Figure 1 The template for $m(d_i, d_o)$ in Imp_{sing}.

As for singular implementations, we use the template given in Fig. 1. Intuitively, there is one non-deterministically assigned thread id (i) which each thread checks whether is equal to its own. There is a local object for each thread, like the isolated implementation template explained above. Additionally, there is another object, $\texttt{O}$, visible to all threads. If the thread with identifier $t \neq i$ invokes a method m with input d_i, then it applies $m(d_i)$ on $\texttt{O}$ atomically (e.g. performing the operation only after acquiring a global lock and releasing upon completion). Otherwise, if $t = i$, then the thread checks whether the sequence it has locally performed so far (kept in the thread-local variable lseq) is robust. If not, it proceeds like other threads, atomically applying the method. If the sequence so far has been robust, the result of applying $m(d_i)$ to it is checked again for robustness. If appending $m(d_i, d_o)$ to lseq leaves it robust, lseq is updated and d_o which is the result of applying $m(d_i)$ to $\mathcal{D}$ after lseq is returned. Otherwise, the sequence up to now is atomically applied to $\texttt{O}$ and t becomes fully synchronized.

5 From SC to Linearizability - Forced Synchronization

The previous section showed that the definition of sequentially consistency is too weak. If it were to be taken as is as the correctness criterion, certain *broken* implementations, such as the isolated or singular implementations, would be correct. We also know that the same implementations are not linearizable. On the one hand the synchronization required for achieving linearizable data structures is also the culprit for non-scalable implementations. On the other hand complete disregard for synchronization allowed by sequential consistency leaves us with pathological implementations. In this section we propose a way to quantitatively bridge the gap from sequential consistency to linearizability.

Our idea is to limit the number of consecutive (total) events that a thread can execute before being forced to synchronize. Let $\vec{\mathbf{h}}$ be a threaded $\mathcal{D}$-history. For any $t \in T$, let e_j be the i^{th} event of t if $\mathbf{h}(t)(i) = e_j$. For any SC threaded $\mathcal{D}$-history $\vec{\mathbf{h}}$, let us call a sequential threaded $\mathcal{D}$-history $\vec{\mathbf{s}}$ a *serialization* of $\vec{\mathbf{h}}$ if for all $t \in T$, $\mathbf{s}(t) = \mathbf{h}(t)$.

▶ **Definition 20** (k-serial)**.** Let a threaded $\mathcal{D}$-history $\vec{\mathbf{h}}$ be SC. Then, $\vec{\mathbf{h}}$ is k-serial if there exists a serialization $\vec{\mathbf{s}}$ of $\vec{\mathbf{h}}$ such that for any $t \in T$, $e, e' \in \mathbf{h}$ whenever e' is the i^{th} event of t and $e' \prec_{\mathbf{h}} e$, then for all $j \leq i - k$ we have $\mathbf{h}(t)(j) <_{\mathbf{s}} e$. An implementation is k-SC if all its traces are k-serial.

Informally, a threaded history is k-serial if a thread cannot continue execution for more than k events without synchronizing with other threads. In SC proper, since there is no explicit requirement for synchronization, for any $k \in \mathbb{N}$ one can construct a threaded $\mathcal{D}$-history such that it is not k-serial as long as $\mathcal{D}$ has at least one robust sequence. In linearizability, this bound is by definition 0; i.e. $\vec{\mathbf{h}}$ is 0-serial iff $\mathbf{h}$ is linearizable. We state these results formally.

▶ **Lemma 21.** *Let $\mathcal{D}$ contain at least one robust sequence.*

- *For any $k \in \mathbb{N}$, there exists a sequence which is $k+1$-serial but not k-serial.*
- *If $\mathbf{h}$ is linearizable, then it is 0-serial.*

Event $d_o = m(d_i)$

```
if self=i then
  d_o ← Obj[self].m(d_i);
  newseq ← lseq·m(d_i,d_o);
  if notRobust(lseq) [∨ cnt ≥ k]
  then
    atomic <commit(lseq)>;
    atomic <d_o ← O.m(d_i)>;
    lseq ← ε; [cnt=0;]
  else
    lseq ← newseq; [cnt++;]
else
  atomic <d_o ← O.m(d_i)>;
return d_o;
```

Figure 2 The template for $m(d_i, d_o)$ in k-SC.

It is straightforward to implement k-SC data structures by modifying the singular implementations given in the previous section (modifications shown by the boxed code of Fig. 2). Events are performed locally without synchronization as long as the the sequence so far has been robust and its length is less than k (the additional disjunct [cnt $\geq k$]). Once either of the conditions is violated, the effects of all events seen so far are committed to the shared data structure. Until then, the local sequence and its length is updated (the increment [cnt++]). Observe that if k is taken to be 0, the additional disjunct will always evaluate to true, forcing synchronization at each call, thereby guaranteeing linearizability.

6 Conclusion

We have shown that sequential consistency despite its appeal is too weak to be used as an alternative to linearizability in specifying concurrent data structure correctness. For almost all well-known data structures, sequentially consistent implementations thereof can have undesirable behavior. For instance, it is possible for a thread in a sequentially consistent queue implementation to observe the queue as empty regardless of what the other threads are doing.

As a first step to bridge the gap between sequentially consistent and linearizable implementations, we also propose a quantitative constraint to capture implementations that lie between the two consistency conditions. In a k-SC implementation, a thread is allowed to proceed without synchronization only for a determined number of consecutive events after which it is required to synchronize.

One possible future work is the development of concrete data structures that are k-SC and investigate the relation between particular values of k and some notion of overall

progress. Another possibility is to check whether a similar strengthening of other consistency conditions either weaker than (memory models of modern processors, such as x86 or ARM) or incomparable to (e.g. quiescent consistency) sequential consistency is useful.

References

1. Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *Proceedings of the 14th International Conference on Principles of Distributed Systems*, OPODIS'10, pages 395–410. Springer-Verlag, 2010.
2. Mustaque Ahamad, Rida A. Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '93, pages 251–260. ACM, 1993.
3. Mustaque Ahamad, Gil Neiger, JamesE. Burns, Prince Kohli, and PhillipW. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.
4. Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The spraylist: A scalable relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 11–20. ACM, 2015.
5. Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: Specification, verification, optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284. ACM, 2014.
6. James R. Goodman. Cache consistency and sequential consistency. Technical Report 1006, Computer Sciences Department, University of Wisconsin, February 1991.
7. Andreas Haas, Thomas A. Henzinger, Christoph M. Kirsch, Michael Lippautz, Hannes Payer, Ali Sezgin, and Ana Sokolova. Distributed queues in shared memory: Multicore performance and scalability through quantitative relaxation. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 17:1–17:9. ACM, 2013.
8. Abdelsalam Heddaya and Himanshu Sinha. Coherence, non-coherence and local consistency in distributed shared memory for parallel computing. Technical report, Computer Science Department, Boston University, 1992.
9. Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative relaxation of concurrent data structures. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 317–328. ACM, 2013.
10. Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.
11. Radha Jagadeesan and James Riely. Between linearizability and quiescent consistency. In *Automata, Languages, and Programming*, Lecture Notes in Computer Science, pages 220–231. Springer Berlin Heidelberg, 2014.
12. ChristophM. Kirsch, Michael Lippautz, and Hannes Payer. Fast and scalable, lock-free k-fifo queues. In *Parallel Computing Technologies*, Lecture Notes in Computer Science, pages 208–223. Springer Berlin Heidelberg, 2013.
13. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, 1979.
14. R.J. Lipton and J.S. Sandbert. Pram: A scalable shared memory. Technical Report TR-180-88, Department of Computer Science, Princeton University, August 1988.
15. Hamza Rihani, Peter Sanders, and Roman Dementiev. Multiqueues: Simpler, faster, and better relaxed concurrent priority queues. *CoRR*, abs/1411.1209, 2014.
16. Nir Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, March 2011.